Modelling Patterns for Analysis and Design of Real-Time Systems

Oana Florescu, Jeroen Voeten, Henk Corporaal

ES Reports
ISSN 1574-9517

ESR-2006-05
15 July 2006

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Modelling Patterns for Analysis and Design of Real-Time Systems[*]

Oana Florescu, Jeroen Voeten and Henk Corporaal

Eindhoven University of Technology
Electrical Engineering Department
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`o.florescu@tue.nl`

Abstract

To ensure correctness and performance of real-time embedded systems, early evaluation of properties is needed. Based on design experience for real-time systems and using the concepts of the POOSL language, we introduce modelling patterns that allow easy composition of models for design space exploration. These patterns cover different types of real-time tasks, resources and mappings, and include also aspects that are usually ignored in classical analysis approaches, like task activation latency or execution context switches. The construction of system models can be done by integrating the necessary patterns, as illustrated in two case studies.

1. Introduction

Complex real-time embedded systems are usually comprised of a combination of hardware and software components that are supposed to synchronise and coordinate different processes and activities. From early stages of the design, many decisions must be made to guarantee that the realisation of such a complex machine meets all the functional and non-functional (timing) requirements.

One of the main problems to address concerns the most suitable architecture of the system such that all the requirements are met. To properly deal with this question, the common approaches are design space exploration and system level performance analysis. An extensive overview of such methodologies is given in [3] and [12]. They range from analytical computation (Modular Performance Analysis [25]) to simulation-based estimation (Spade [16], Artemis [21]). These are often specialised techniques which claim that general purpose languages are ill-suited for system-level analysis. However, due to the heterogeneity and complexity of systems, for the analysis of different aspects different models need to be built and their coupling is difficult. Therefore, a unified model, covering all the interesting aspects, is actually needed to speed up the design process. This is how the Unified Modelling Language (UML) [20] came to be conceived. The language was designed mainly for object-oriented software specification, but recently it was extended (UML 2.0) to include (real-time) systems as well.

During the development of new systems, specific problems are encountered again and again, and experienced designers apply the solutions that worked for them in the past [9]. These pairs of problem-solution are called *design patterns* and their application helps in getting a design “right” faster. With the increase in the development of real-time systems, design patterns were needed for dealing with issues like concurrency, resource sharing, distribution [8]. As UML has become the standard language for modelling, these patterns are described in UML. However, the semantics of the language is not strong enough to properly deal with the analysis of *real-time* system

[*] This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

behaviour. Therefore, an expressive and formal modelling language is required instead in order to capture in a compact model *timing*, *concurrency*, *probabilities* and *complex behaviour*.

Design patterns refer to problems encountered in the design process itself, but problems appear also in the specification of components that are commonly encountered in complex systems [11]. Although components of the analysed systems exhibit some common aspects for all real-time systems (e.g. characteristics of tasks like periodicity or aperiodicity, processors, schedulers and their overheads), they are built everytime from scratch and similar issues are encountered over and over.

Contributions of the paper. To reduce the amount of time needed to construct models for design space exploration, we propose *modelling patterns* to easily compose models for the design space exploration of real-time embedded systems. These modelling patterns, provided as a library, act like templates that can be applied in many different situations by setting the appropriate parameters. They are based on the concepts of a mathematically defined general-purpose modelling language, POOSL [24], and they are presented as UML diagrams. These boilerplate solutions are a critical success factor for the practical application in an industrial setting and are a step towards the (semi-) automated design space exploration in the early phases of the system life-cycle.

This paper is organised as follows. Related research work is presented in Section 2. Section 3 briefly presents the POOSL language, whereas Section 4 provides the modelling patterns. The composition of these patterns into a model is discussed in Section 5 and their analysis approach in Section 6. The results of applying this approach on two case studies are presented in Section 7. Conclusions are drawn in Section 8.

2. Related Research

An extensive overview of performance modelling and analysis methodologies is given in [3] and [12]. They range from analytical computation (Modular Performance Analysis [25], UPPAAL [4]) to simulation-based estimation (Spade [16], Artemis [21]). The techniques for analytically computing the performance of a system are exhaustive in the sense that *all* possible behaviours of the system are taken into account. On the other hand, simulation of models allows the investigation of a *limited* number of all the possible behaviours of the system. Thus, the obtained analysis results are *estimates* of the real performance of the system. To obtain credible results, both types of techniques require the models created to be amenable to mathematical analysis (see [23]), using mathematical structures like Real-Time Calculus [7], timed automata [2] or Kahn process networks [13]. As in general analytical approaches do not scale with the complexity of the industrial systems, simulation-based estimation of performance properties is used more often. In this context, the estimation of performance is based on statistical analysis of simulation results.

With respect to timing behaviour, an impressive amount of work has been carried out in the area of schedulability analysis for meeting hard real-time requirements (e.g. [17], [6], [5]) focussing on worst case. However, less work addresses the analysis of systems with probabilistic behaviour. For soft real-time systems, it is important to analyse the variations in the runtime behaviour to determine the likelihood of occurrence of certain undesired situations and, based on that, to dimension the system. In [18] and [23] it is showed that the techniques proposed in this area are quite restrictive. Some of them target certain application classes, being limited to uni-processor architectures or supporting only exponential distributions for expressing the probabilistic behaviour; other approaches address specific scheduling policies or assume highly-loaded systems.

To overcome these issues, in this paper we present a modelling approach that can capture any kind of probabilistic distribution of system behaviour. Moreover, for the analysis of timing behaviour any scheduling policy is allowed. Although the evaluation of the system properties is based on simulations, due to the formal semantics of the language, the accuracy of the results can be determined.

3. POOSL modelling language

The Parallel Object-Oriented Specification Language (POOSL) [24] lies at the core of the Software/Hardware Engineering (SHE) system-level design method. POOSL contains a set of powerful primitives to formally describe concurrency, distribution, synchronous communication, timing and functional features [22] of a system into a single executable model. Its formal semantics is based on timed probabilistic labelled transition systems [15]. This mathematical structure guarantees a unique and unambiguous interpretation of POOSL models. Hence, POOSL is suitable for specification and, subsequently, verification of correctness and evaluation of performance for real-time systems.

POOSL consists of a *process* part and a *data* part. The process part is used to specify the behaviour of active components in the system, the processes, and it is based on a real-time extension of the Calculus of Communicating Systems (CCS) [19]. The data part is based on traditional concepts of sequential object-oriented programming. It is used to specify the information that is generated, exchanged, interpreted or modified by the active components. As mostly POOSL processes are presented in this paper, fig. 1 presents the relation between the UML class diagram and the POOSL process class specification. The name compartment of the class symbol for process classes is stereotyped with `<<process>>`. The attributes are named `<<parameters>>` and allow parameterising the behaviour of a process at instantiation. The behaviour of a process is described by its `<<methods>>` which may include the specification of sending (**!**) and/or receiving (**?**) of `<<messages>>`[1].

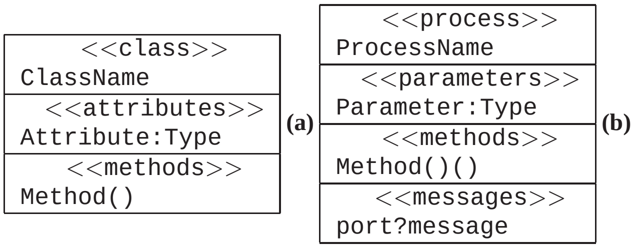


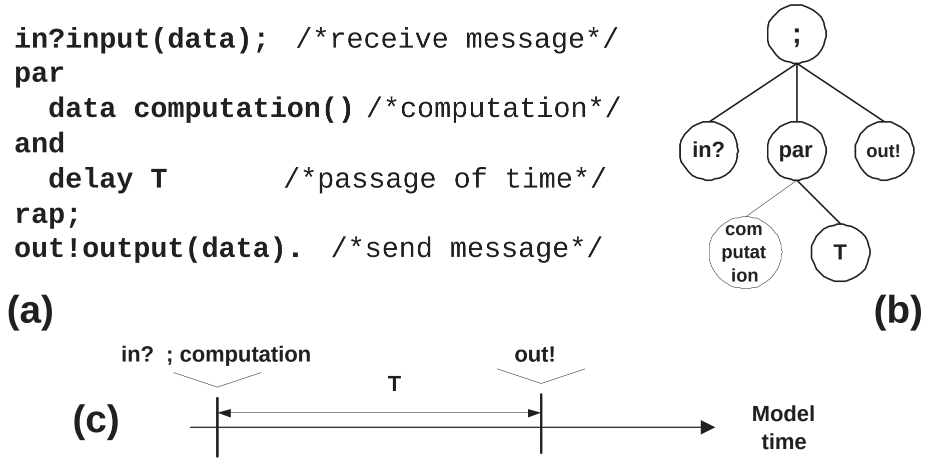
Figure 1. UML (a) vs. POOSL process (b) class specification

The SHE method is accompanied by two tools, SHESim and Rotalumis. SHESim is a graphical environment intended for incremental specification, modification and validation of POOSL models. Rotalumis is a high-speed symbolic execution engine[2], enabling fast evaluation of system properties. Compared with SHESim, Rotalumis improves the execution speed by a factor of 100 by compiling the model into an intermediate format before executing it.

The algorithm residing at the core of both tools for symbolic execution of a model has been proven correct in [10] with respect to the formal semantics of the language. Each POOSL specification is automatically translated into Process Execution Trees (PETs) (see fig. 2a and fig. 2b respectively). A PET represents the remaining behaviour of a POOSL process. The leaves of the tree are statements describing the timed behaviour of that process, whereas the internal nodes represent compositions of their children (e.g. parallel, sequential, nondeterministic choice). There are two phases during execution. First, a PET scheduler asynchronously grants all eligible atomic actions, such as communications or data computations, without any passage of time, until there are no other actions available at the current moment. Then, time passes synchronously for all PETs, until the moment when an action becomes eligible again and the first phase is resumed. The internal state of each PET is changed according to the choices made by the scheduler and a timed trace is maintained for later analysis (fig. 2c). As there are potentially infinitely many paths, simulation completeness cannot be claimed, in general, because exhaustive exploration is an NP-complete problem.

[1] More details about the UML profile for POOSL can be found in [23].

[2] We use the word execution to denote simulation throughout the paper.



```
in?input(data);  /*receive message*/
par
  data computation() /*computation*/
and
  delay T       /*passage of time*/
rap;
out!output(data).  /*send message*/
```

Figure 2. Example of a POOSL model

4. Modelling patterns

One of the approaches for performing systematic design space exploration is the Y-chart scheme (fig. 3), introduced in [14]. This scheme makes a distinction between applications (the required functional behaviour) and platforms (the infrastructure used to perform this functional behaviour). We have added to this scheme the model of the environment that is to be controlled by the system through its functionality. Although physically the environment is connected to the platform, logically it is connected to the application that controls it and thus it was placed accordingly in the scheme. The design space can be explored by evaluating different mappings of applications onto platforms.

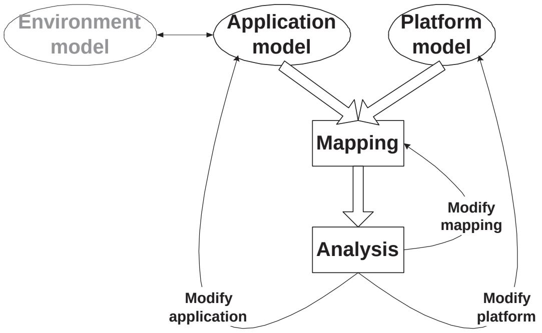


Figure 3. Y-chart scheme

As real-time embedded systems usually contain components with common characteristics, like tasks, computation / communication resources, modelling patterns can be developed such that when another model of the same or of a similar system needs to be built, the appropriate patterns and their parameters can be chosen and used immediately.

Table 1 presents the modelling patterns developed and used in the case studies presented in the paper. The application model is described through real-time tasks, which are characterised by deadline, load (which represents the number of instructions that the task needs to execute at a certain activation and which is determined based on best-case/worst-case load and a certain load distribution), latency of task activation, plus period and number of iterations for periodic tasks. In Section 4.1, these patterns are presented and their parameters explained. The platform model consists of (computation and/or communication) resources, which are uniformly characterised by

Table 1. Modelling patterns

Y-chart part	Pattern Name	Parameter Name
Application Model	PeriodicTask	period (T) deadline (D) BCload WCload loadDistribution latency (l) iterations
	AperiodicTask	deadline (D) BCload WCload loadDistribution latency (l)
Platform Model	Resource	initial latency throughput
	Scheduling	scheduling policy
Environment Model	Environment	arrival stream upper bound (u) lower bound (l)

an initial latency and throughput, and the scheduling policies that handle the concurrent requests (see Section 4.2). The mapping stage of the Y-chart is explained in Section 4.3, whereas the model of the environment, characterised by an event stream with a certain distribution of arrival between an upper and a lower bound, in Section 4.4.

4.1. Application model

The functional behaviour of a real-time embedded system is implemented through a number of tasks that may communicate with each other. Task activation requests can be periodic (time-driven), being activated at regular intervals equal to the task period *T*, or aperiodic (event-driven), waiting for the occurrence of a certain event. There are three types of uncertainties, shown in fig. 4, that may affect a task:

- activation latency: caused, for example, by the inaccuracies of the processor clock that might drift from the reference time because of temperature variations. For event-driven tasks, the performance of the runtime system, which cannot continuously monitor the environment for events, may also have influence.

- release jitter: caused by the interference of other tasks that, depending on the scheduling mechanism, may impede the newly activated task to start immediately its execution

- output jitter: caused by the cumulated interference of other tasks in the system, the scheduling mechanism that may allow preemption of the executing task, the variation of the activation latency and even of the execution time of the task itself, which may depend on the input data.

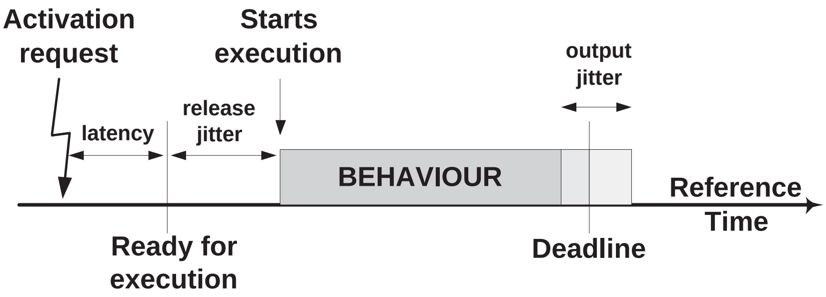


Figure 4. Real-time task parameters

In classical real-time scheduling theory [6], the release jitter and, to some extent, the output jitter[3] can be computed, but the activation latency is ignored. Therefore, modelling patterns are provided here to overcome this problem (see fig. 5 for the UML diagrams and fig. 6 for the POOSL specification). The core of these patterns is the complete decoupling of the desired timing behaviour from the actual timing when the behaviour of the task executes. For a time-driven task (see fig. 6a), the **par-and-rap** POOSL statement indicating parallel composition in PERIODIC, is used to decouple the task period from its real activation moment. The **par** branch is used to execute the actual BEHAVIOUR, possibly with latency, while the **and** branch is used to determine the next period by delaying exactly *T* and then recursively calling itself. The actual deadline of the task is given as parameter of the BEHAVIOUR because it is considered with respect to the reference time: its value depends on the amount of latency (D+*l*-lat). Furthermore, the execution of the periodic task is modelled to be finite (if *iterations* > 0) or infinite (if *iterations* $= -1$).

```
<<process>>
PeriodicTask
<<parameters>>
T:Real
D:Real
BCload:Integer
WCload:Integer
loadDistribution:Distribution
l:Real
iterations:Integer
<<methods>>
Init()()
Periodic()()
Behaviour()()
<<messages>>
```

```
<<process>>
AperiodicTask
<<parameters>>
D:Real
BCload:Integer
WCload:Integer
loadDistribution:Distribution
l:Real
<<methods>>
Init()()
Aperiodic()()
Behaviour()()
<<messages>>
in?event
out!output
```

Figure 5. UML task patterns

The data object *Latency* is an instance of a class representing a discrete uniform distribution in $[0, l]$. In the ideal situation, $l = 0$ and *sample* always returns zero. If $l > 0$, *sample* returns a value in $[0, l]$ and the actual activation moment drifts from the reference time with $\pm l$. Hence, BEHAVIOUR is invoked somewhere in the interval $[0, l] \cup [n * T - l, n * T + l], l < T, n \in \mathcal{N}^{+}$. Note that each time BEHAVIOUR is called, it is possible that the previous activation is still in progress. During simulation of the model, the designer can be informed if two or more activations are in progress at the same time or if any task misses its deadline.

[3] The output jitter can be computed without taking into account possible variations nor dependencies on the input data.

```
PERIODIC()() |lat : Real|
if (iterations != 0) then
 par
  delay T-l;
  lat := 2*Latency sample();
  delay lat;
  BEHAVIOUR(D+l-lat)()
 and
  delay T;
  if (iterations != -1) then
   iterations := iterations-1 fi;
  PERIODIC()()
 rap
fi.
```

(a) time-driven task

```
APERIODIC()() |lat : Real|
 in?event;
 par
  par
   lat := Latency sample()
   delay lat;
   BEHAVIOUR(D-lat)()
  and
   delay D
  rap;
  out!output
 and
  APERIODIC()()
 rap.
```

(b) event-driven task

Figure 6. POOSL task patterns specification

The event-driven tasks are activated at the arrival of a message *event* on the port *in* (fig. 6b). For this reason, there is no need to express a certain number of iterations if the execution of an aperiodic task is not infinite, as it is blocked/stopped anyway waiting for an event to happen. Usually, an aperiodic task is required to output its computations result (*out!output*) before some deadline *D*. If BEHAVIOUR does not finish by that time, the output is postponed, causing output jitter. During simulation, the designer is informed about such situations.

In a real-time system, the functional behaviour of a task consists of independent computations and inter-task communications. These two aspects can be intuitively captured in the BEHAVIOUR method specification (see for example fig. 7). While the modelling patterns for tasks can be directly used, by instantiating objects of the appropriate class and setting the necessary values for the parameters, the specification of BEHAVIOUR must be overloaded by the user at design time.

```
BEHAVIOUR(deadline : Real)() |tstart, tstop : Real|
 tstart := currentTime;
 COMMUNICATE(messageLength)();
 tstop := currentTime;
 COMPUTE(deadline - tstop + tstart)().
```

Figure 7. BEHAVIOUR **model**

4.2. Platform model

The platform on which the software runs can be described as a collection of resources. A resource is able to provide the capacity to perform the desired functional behaviour. The modelling patterns provided here allow a unified way of modelling resources by exploiting their common characteristics. There is no large conceptual difference between a CPU and a bus: they both receive requests, execute them and send back a notification on completion.

If a resource is shared by a number of concurrent tasks, a scheduler is needed to arbitrate the access to the resource. Depending on the type of application and the resources, an appropriate scheduling algorithm can be modelled: preemptive, non-preemptive, priority-based, earliest deadline first, etc. A general preemptive scheduling behaviour, whose UML class diagram is given in fig. 8, can be modelled as a POOSL process, as shown in fig. 9. The core of this pattern relies on the non-deterministic choice that allows any possible sequence in the scheduler behaviour. It can either receive scheduling requests from newly activated tasks (the outer **sel** branch), or notifications from the platform about completed requests (the **or** branch). The newly activated request is put in the

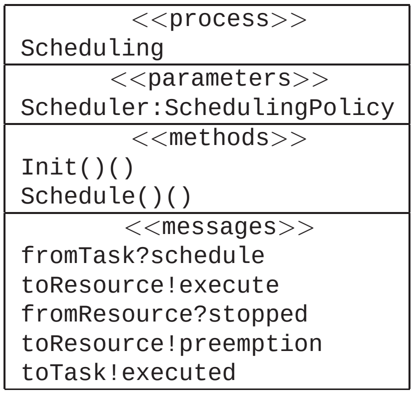


Figure 8. UML scheduling pattern

```
SCHEDULE()() | req, oldreq : Request |
 sel
   fromTask?schedule(req);
   Scheduler scheduleRequest(req);
   if (Scheduler hasHighestPriority(req) == true) then
      sel
        toResource!execute(req)
      or
        toResource!preemption;
        fromResource?stopped(oldreq);
        toResource!execute(req);
        Scheduler update(oldreq)
      les
   fi;
   SCHEDULE()()
 or
   fromResource?stopped(oldreq);
   toTask!executed;
   req := Scheduler removeRequest(oldreq);
   if (req != nil) then toResource!execute(req) fi;
   SCHEDULE()()
 les.
```

Figure 9. POOSL scheduling pattern specification

list of scheduled requests by calling the data method *scheduleRequest(req)*. If *req* has the current highest priority, it is sent to the resource for being immediately handled (the inner **sel** branch). As the resource might already be running another request, the corresponding **or** branch models the situation when the old request is preempted and rescheduled (*update(oldreq)*). In the outer **or** branch, the scheduler receives completed requests from the resource and removes them from the ready list by calling *removeRequest(oldreq)*, which also returns the next scheduled request, if there is one.

The data object *Scheduler* is an instance of a class that implements a scheduling algorithm. Different subclasses of the *SchedulingPolicy* abstract class (fig. 10) may implement different algorithms (e.g. EDF, RMA) and for each resource, a different scheduler can be instantiated. It can be changed anytime during the design, without affecting the rest of the model. The methods of this class require as parameter a data object of type *Request*, containing the information needed for scheduling: release time, load (instructions number of a task / length of a message), and deadline. Such a data object is built at run-time, during model execution, as the scheduler need not make any difference whether it is a task or a message to schedule on the underlying resource. To model a non-preemptive scheduler, the method *hasHighestPriority*(req) should return *false* if there is a task already being executed. Note that, whenever a new instance of a task is released, it is scheduled, without taking into account that a previous instance might still be running. If this situation is unwanted, it can be detected and reported. In case the deadline of a request is missed, the scheduler detects it when it removes the request from the list, and announces this as an error during simulation.

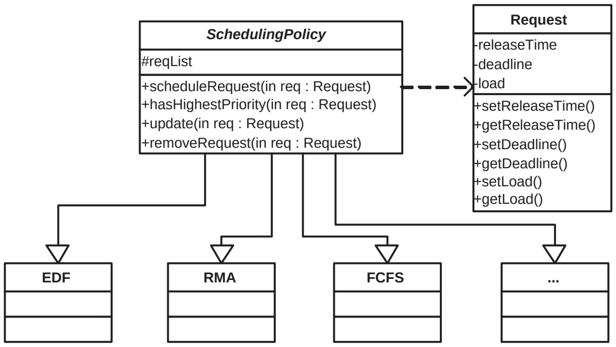
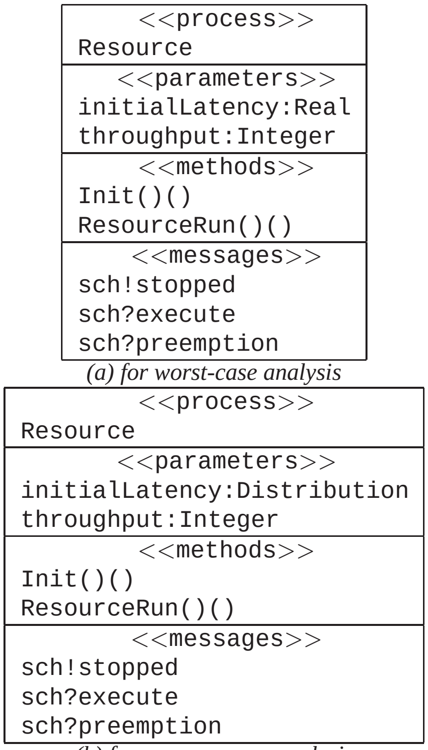


Figure 10. UML diagram for scheduling policies



(a) for worst-case analysis

(b) for average-case analysis

Figure 11. UML resource patterns

Fig. 12 presents the resource model as a POOSL process receiving execution requests from the scheduler. Before the actual execution, the resource has an initial latency, which is given as a parameter of the modelling pattern, as shown in the UML diagrams in fig. 11. For a worst-case analysis of the system, a fixed, worst-case value of it is provided, whereas for an average case analysis, it is given as a distribution. The initial latency is justified by: in case of a CPU, the context switch that proceeds the execution of a newly scheduled task for saving the status of the previous task and loading the current task; in case of a bus, the time it takes for the first bit of the message to be transferred, which depends mostly on the communication protocol used. After the initial delay, the resource lets the time pass according to the execution time associated to the request. The execution time is computed based on the load of the request (representing either the number of instructions of a task or a message length) and the *throughput* of the resource, which is the second parameter. The core concept behind the presented modelling pattern for a resource is the possibility of the language to express the breaking of the execution, needed if the scheduling mechanism allows preemption. In POOSL, this can be modelled with the **abort** statement. The remaining execution time of the request (actually the remaining load) is computed and updated (*req setLoad(loadLeft)*) and the request is sent back to the scheduler. Preemption is usually the case for computation resources, and less common for communication. Nevertheless, as preemptions and their associated latencies (like context switches) might have a large influence on the finishing time and the output jitter of a task, they must be taken into account.

```
RESOURCERUN()() | req: Request, loadLeft, tstart, tstop : Integer |
  sch?execute(req);
  delay initialLatency;
  tstart := currentTime;
  abort
    delay req getLoad() / throughput
  with sch?preemption;
  tstop := currentTime;
  loadLeft := req getLoad() - (tstop - tstart) * throughput;
  req setLoad(loadLeft);
  sch!stopped(req);
  RESOURCERUN()().
```

(a) for worst-case analysis

```
RESOURCERUN()() | req: Request, loadLeft, tstart, tstop : Integer |
  sch?execute(req);
  delay initialLatency sample();
  tstart := currentTime;
  abort
    delay req getLoad() / throughput
  with sch?preemption;
  tstop := currentTime;
  loadLeft := req getLoad() - (tstop - tstart) * throughput;
  req setLoad(loadLeft);
  sch!stopped(req);
  RESOURCERUN()().
```

(b) for average-case analysis

Figure 12. POOSL resource pattern specification

4.3. Mapping model

To analyse the performance of a system, a mapping of the application model onto the platform model is composed (fig. 13). In this approach, an explicit mapping has been chosen, represented by a POOSL communication channel linking one or more tasks to a resource. A task mapped onto a resource is able to send execution requests to that resource, modelled as POOSL messages sent through the mapping channel. For example, when the task needs to COMPUTE (as depicted in fig. 14), a message is sent to the CPU containing the required deadline and the imposed load (*toRes!execute(deadline, load)*). Such a message is encapsulated in a data object of type *Request*, discussed in the previous section, which arrives at the CPU scheduler, where it is first scheduled and then sent to execution. As soon as the execution is finished, the task is informed to continue its behaviour (*fromRes?executed*).

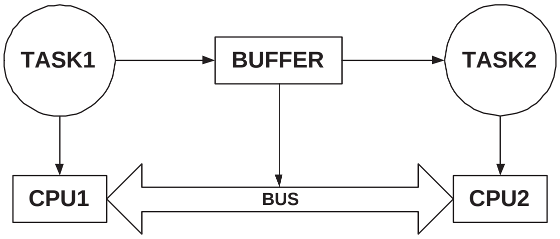


Figure 13. Mapping model

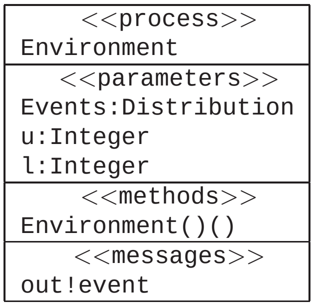
```
COMPUTE(deadline : Real)()
  toRes!execute(deadline, loadDistribution sample());
  fromRes?executed.
```

Figure 14. COMPUTE **model**

Although POOSL channels can model inter-task communication, they cannot be mapped onto communication resources; thus, a buffer model is required to completely decouple the application from the platform model. When a task needs to COMMUNICATE with another task over a bus (fig. 7), the message is put in the buffer. The buffer sends a request to the bus to transfer *messageLength* bytes and waits for its completion. By simply connecting tasks to resources in different ways, easy exploration of different mappings can be achieved.

4.4. Environment model

Research in the area of classical scheduling theory mainly focussed on the assumption that all external events arrive either perfectly periodic or aperiodic, based on a predefined arrival pattern, without any latencies or sporadic effects. Therefore, only models of the application and the platform were typically considered for reasoning about the properties of the system.



```
ENVIRONMENT()()

Events:=new(Distribution)
 ofType(Uniform)
 withParameters(l, u);
while (true) do
  delay Events sample();
  out!event
od.
```

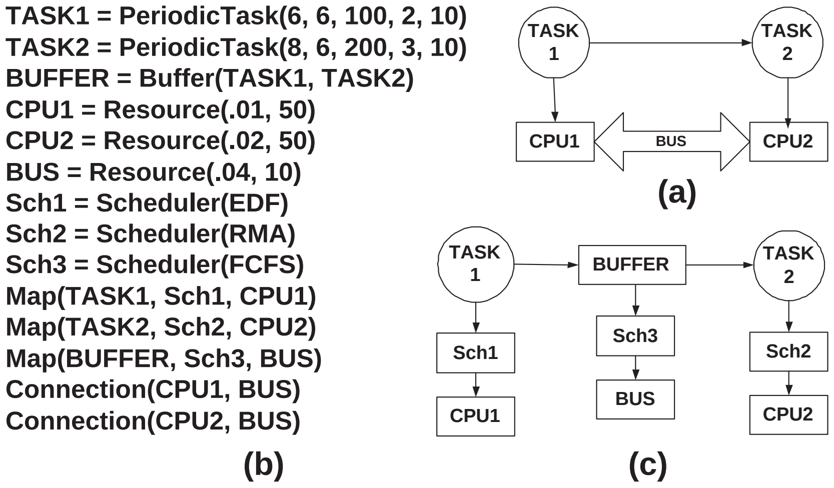
Figure 15. Environment pattern and specification

However, to reason accurately about the properties of an embedded system, its whole behaviour should be modelled realistically, including a probabilistic model of the environment that triggers the events. For this purpose, a discrete-event approximation of the continuous-time behaviour of the physical components can be modelled in terms of event streams occurring according to some distribution. An example of such a model, generating a stream based on a uniform distribution, is given in fig. 15.

5. Model composition from patterns

To build a model of a real-time system for design space exploration, its specific components that correspond to the modelling patterns described in the previous section must be identified together with their parameters. The names of the necessary patterns and their parameters, together with the specification of the mapping (which task is scheduled on which processor, etc.) and the layout of the platform (which processor is connected to which bus) can be provided as the configuration of the system. From such a configuration, the POOSL model of the system can be automatically generated and fed to SHESim or Rotalumis tools for analysis. As an example, for the system in fig. 16a, the specification of the necessary patterns may look like the one in fig. 16b, and the structure of the generated model is shown in fig. 16c.

For design space exploration, different configurations must be compared. To do this, changes in the initial configuration may be done and the POOSL model re-generated in order to analyse them. To specify a different



```
TASK1 = PeriodicTask(6, 6, 100, 2, 10)
TASK2 = PeriodicTask(8, 6, 200, 3, 10)
BUFFER = Buffer(TASK1, TASK2)
CPU1 = Resource(.01, 50)
CPU2 = Resource(.02, 50)
BUS = Resource(.04, 10)
Sch1 = Scheduler(EDF)
Sch2 = Scheduler(RMA)
Sch3 = Scheduler(FCFS)
Map(TASK1, Sch1, CPU1)
Map(TASK2, Sch2, CPU2)
Map(BUFFER, Sch3, BUS)
Connection(CPU1, BUS)
Connection(CPU2, BUS)
```

Figure 16. Use of patterns

mapping, the **`Map`** specifications must be changed according to the new task-to-resource mapping. To change the architecture components, simply change the **`Resource`** specifications and/or their parameters. Similarly, the layout of the platform can be changed in the **`Connection`** specification tags. In this way, the model can be easily tuned to specify different possibilities in the design space without any knowledge about the underlying formal model that will be generated in accordance with the description of the new configuration.

6. Model analysis

By composing together the necessary modelling patterns as shown in Section 5, the complete model of a system can be built and validated. For each configuration specified and generated, during the execution of the model, the scheduler can report if there are any tasks that miss their deadlines. Furthermore, based on the POOSL semantics derived from CCS, it can be detected if there is any deadlock in the system. If all the deadlines are met and there is no deadlock, then the corresponding architecture is a good candidate that meets the system requirements.

However, for soft real-time systems, it is allowed that some deadlines are missed (usually there is a requirement for an upper limit). Therefore, in this case, it is especially useful that the analysis of the model can handle and record tasks with multiple active instantiations that have missed their deadlines. The percentage of deadlines missed can be monitored and checked against the requirements if, according to this criterion, the underlying platform is suitable.

Furthermore, as shown in Section 4, the task models are designed relative to a reference time, not to the platform time. This differs from traditional approaches as the performance of the architecture or the drifts of a processor clock do not influence the timeliness of the control of the physical components in the environment anymore. As the environment “runs” relative to the reference time, the designer is able to check if, under different circumstances, the behaviour still meets the critical deadlines.

To correctly dimension a system (the required CPUs performance and buses) such that it works in any situation, the worst-case behaviour of the system must be analysed. This usually means to consider the worst-case execution times for all the activities in the system. On the other hand, the analysis of the average behaviour, based on probabilities, is also important, as it gives a measure of the suitability of the design. If the dimension of the system, needed for the worst-case situation that appears only once in a while, is far bigger than the one needed in average, that could give useful hints for a re-design (e.g. split tasks into smaller ones in order to spread the load onto different CPUs).

Some other useful results the analysis of the proposed model can provide are the release jitter, the output jitter and the number of instances of a task active at the same time.

7. Case studies

In this section, two case studies are presented for which design space exploration has been performed using the modelling patterns proposed in this work. The characteristics of the systems and the results of their analysis follow.

7.1. A printer paper-path

The first case study is inspired by a system architecture exploration for the control of the paper-path of a printer. The high-level view of the system model, visualised using SHESim tool, is given in fig. 17. User's printing requests arrive at the high-level control (HLC) of the machine which computes which activities need to take place when to accomplish the request. The HLC tasks activate the tasks representing the low-level control (LLC) of the physical components of the paper path, like motors, sensors and actuators. As HLC tasks are soft real-time, whereas LLC tasks (fig. 18) are hard real-time, a rather natural solution was to consider a distributed architecture. LLC can be assigned to dedicated processor(s) and connected through a network to the general-purpose processor that runs HLC.

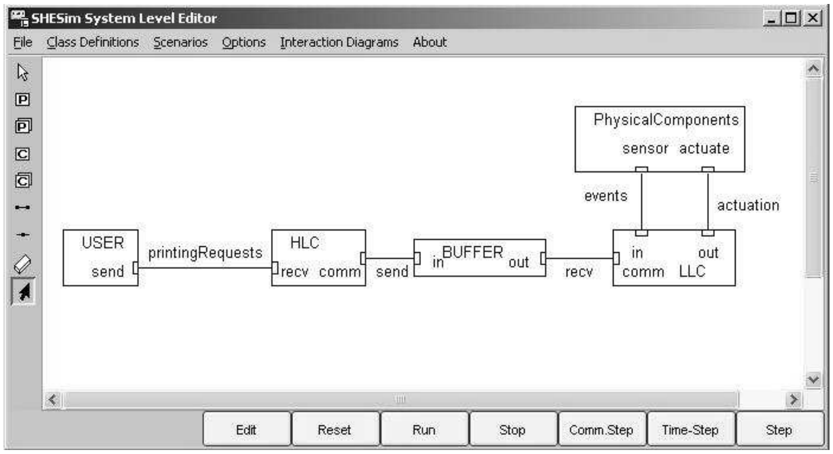
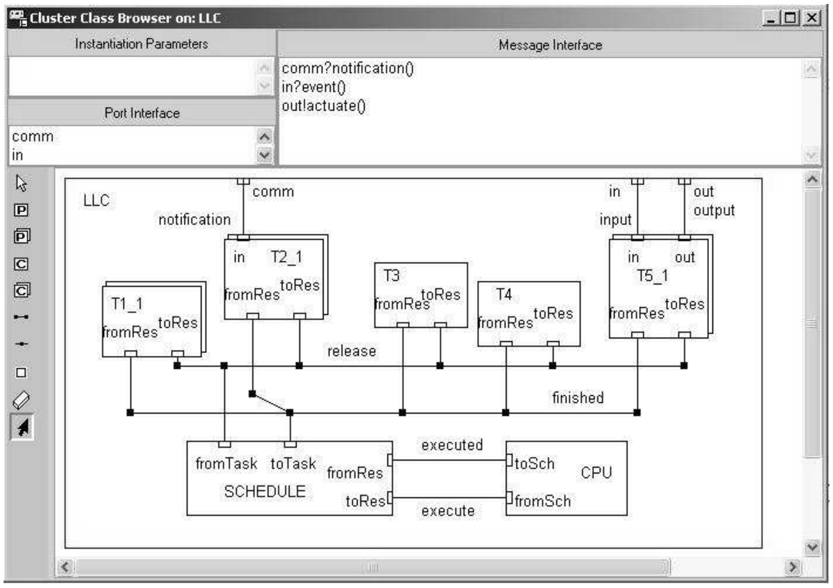


Figure 17. High-level printer control POOSL model

Under these circumstances, the *problem* was mainly *to find an economical architecture for LLC*, whose task parameters are shown in table 2. For the models of the time-driven tasks of type T1, T3 and T4, we took into account a latency of upto 10% of their period. Although tasks of type T2 are activated based on notifications from HLC, they behave completely periodic until the next notification arrives. Therefore, their dynamical behaviour was captured using an aperiodic task which triggers a periodic task with a finite number of activations. Tasks of type T5 are event-driven; therefore, a model of the environment was needed (*PhysicalComponents*), for which we considered event streams with a uniform distribution in $[1, 20]$ ms.

Given the frequency of events and the task execution times, we have analysed three commercially available low-end processors, a 40 MIPS, a 20 MIPS and a 10 MIPS, and compared their utilisations under different schedulers. Fig. 19 presents the results obtained using the earliest deadline first scheduling algorithm. Although the 10 MIPS



Message Interface:
comm?notification()
in?event()
out!actuate()

Port Interface:
comm
in

Figure 18. POOSL LLC model

Table 2. LLC task parameters

Task type	No. of Instantiations	Load	T (ms)	D (ms)
T1	3	3200	2	2
T2	8	1200	2	2
T3	1	2000	2	2
T4	3	800	0.66	0.1
T5	4	160	-	0.064

processor seems to be used the most efficiently (close to its maximum capacity), the analysis of the model showed that some of the deadlines are missed; thus this processor is not a good candidate. For the other two, all deadlines are met and there were no deadlocks detected in the system. Due to the fast execution engine Rotalumis, tens of hours of system behaviour could be covered in less than one minute simulation. Moreover, the analysis of the model gave the values of the maximum release jitter, respectively output jitter of the tasks (for the 20 MIPS they are shown in table 3) which could be checked against the expected margins of errors of the environment control design.

7.2. An in-car navigation system

The second case study is inspired by a distributed in-car navigation system [1]. The system, depicted in fig. 20, has three clusters of functionality: the man-machine interface (MMI) handles the interaction with the user; the navigation functionality (NAV) deals with route-planning and navigation guidance; the radio (RAD) is responsible for basic tuner and volume control, as well as receiving traffic information from the network. For this system, three application scenarios are possible: the ChangeVolume scenario allows users to change the volume; the ChangeAddr scenario enables route planning by looking up addresses in the maps stored in the database; in the HandleTMC scenario the system needs to handle the navigation messages received from the network. Each of

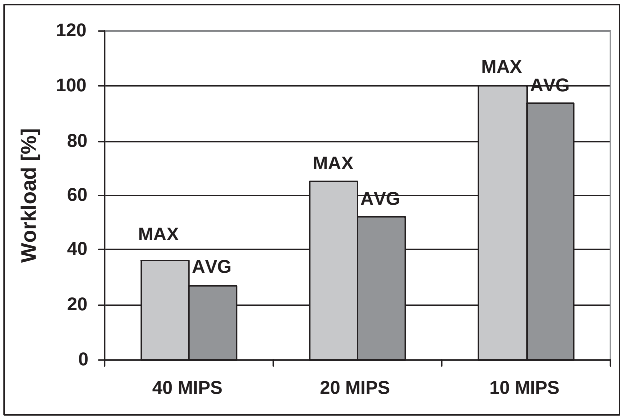


Figure 19. CPU workload comparison

Table 3. Tasks jitter for the 20 MIPS

Task type	Release jitter (ms)	Output jitter (ms)
T1	0.466	1.852
T2	0.466	1.852
T3	0.414	1.884
T4	0.042	0.128
T5	0.472	1.094

these scenarios is described by a UML message sequence diagram, like the one shown in fig. 21. A detailed description of the system and of its scenarios can be found in [25].

The *problem* related to this system was *to find suitable platform candidates* that meet the timing requirements of the application. To explore the design space, a few platforms, presented in fig. 22, were proposed and analysed using Modular Performance Analysis (MPA) in [25]. MPA is an analytical technique in which the functionality of a system is characterised by the incoming and outgoing event rates, message sizes and execution times. Based on Real-Time Calculus, hard upper and lower bounds of the system performance are computed. However, these bounds are in general not exact, meaning that they are larger/smaller than the *actual* worst/best case. Thus, the analysis performed is conservative. As the in-car navigation is a soft real-time system that allows a certain percentage of deadline misses, it is doubtfully interesting to explore if there is an architecture of lower cost and performance than what have been obtained with MPA that can still meet the timing requirements.

7.2.1 Worst-case analysis

The UML diagrams specifying the case study provide the worst-case values of the load (number of instructions) imposed by tasks on the CPUs. They also specify what is the rate of task activations (how often the events are triggered) which depends on the scenario in which they appear. Based on these activation rates, priorities were assigned to tasks according to the rate monotonic approach. The timing requirements of the system are also specified in the UML diagrams as end-to-end deadlines for each scenario. The loads of the tasks, the frequencies (f) of activations[4] per scenario and the timing requirements are given in table 4.

[4] Tasks are triggered by the events in the environment as knob turning or messages from the network. In this analysis, the events are assumed to arrive periodic, so the values of the lower and upper limits of the arrival stream in the environmental model are equal.

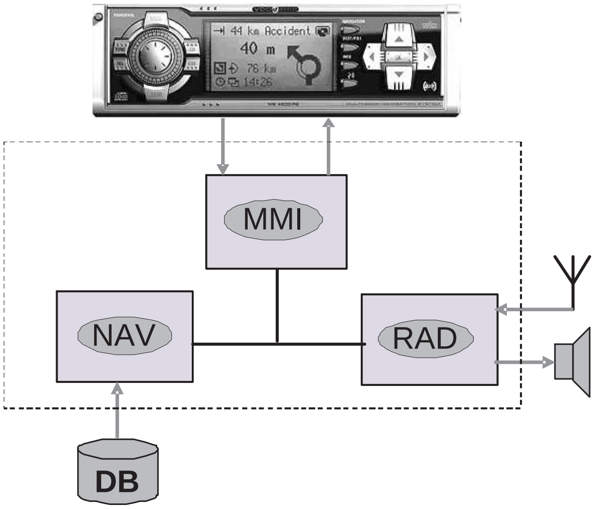


Figure 20. In-car navigation system

Table 4. Timeliness requirements of the system

Scenario name	Deadline [*ms*]	Task name	Load [*instructions*]	f [*1/s*]
ChangeVolume	200	HandleKeyPress	1E5	32
		AdjustVolume	1E5	32
		UpdateScreen	5E5	32
ChangeAddr	200	HandleKeyPress	1E5	1
		DatabaseLookup	5E6	1
		UpdateScreen	5E5	1
HandleTMC	1000	ReceiveTMC	1E6	1/3
		DecodeTMC	5E6	1/3
		UpdateScreen	5E5	1/30

By simulating[5] the behaviour of the system, using each of the proposed architectures in fig. 22, the end-to-end delays were monitored. Fig. 23 shows, as an illustration, the maximum end-to-end delay obtained for HandleTMC scenario when running alone on each of the proposed platforms (from A to E).

The most interesting situations to monitor were the ones in which two scenarios are running in parallel as such a situation can lead to a larger value for the end-to-end delay. In our simulation, we have observed that all the deadlines are met on all the architectures. As an example, the results obtained for different combinations of scenarios on architecture A are presented in table 5. Next to them, the results obtained using MPA and UPPAAL techniques are also provided. Architecture A was chosen for further discussion because it was the one chosen for deeper analysis by both techniques.

MPA is an analysis technique which finds hard upper bounds, not necessarily the actual worst case reached by the model. This explains the larger values that are obtained by applying this method. On the other hand, the results computed by UPPAAL are exact values of the worst case end-to-end delay. It is interesting to observe that our results are very close to UPPAAL (∼1% difference which also represents the accuracy of the results), except

[5] Note that, the simulation was run with the fast execution engine Rotalumis; thus, a few minutes of system simulation represent several hours of runtime behaviour. The simulation was run until an accuracy of 99% of the results was reached.

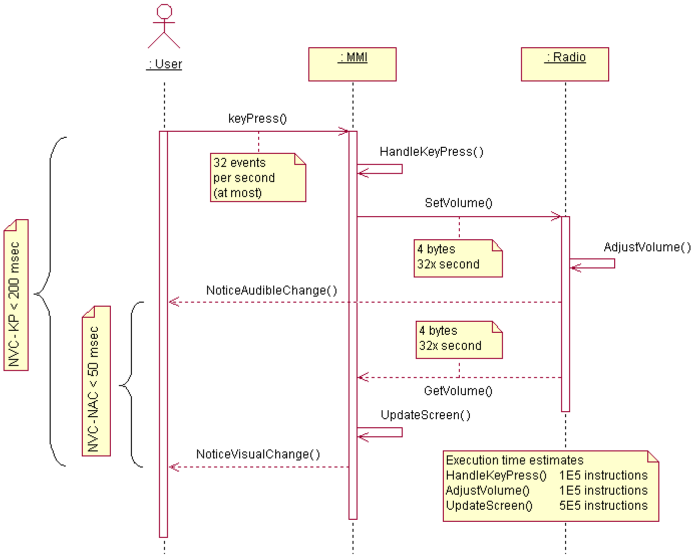


Figure 21. ChangeVolume scenario

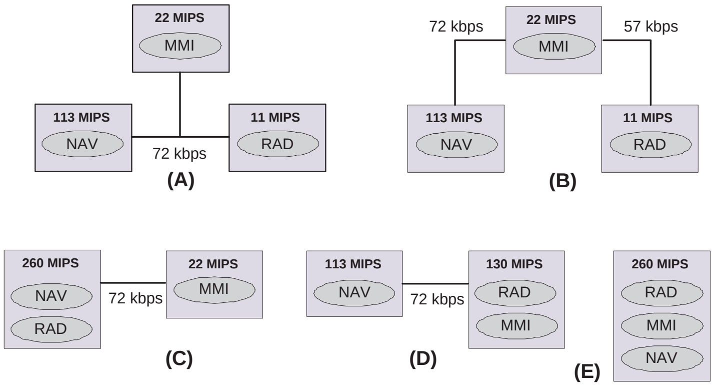


Figure 22. Platforms proposed for analysis

for HandleTMC scenario for which the difference is 7%. For this situation we suspect a miss-match between the corresponding models and this aspect is still under investigation.

Besides keeping track of the end-to-end delays, during simulation, we have also monitored the resources utilisation. For architecture A, the obtained results are presented in table 6. Based on the amount of idle time of the CPUs and on the fact that the worst case values of the delays are much smaller than the specified deadlines, we concluded that the performance of the underlying architecture could be reduced in order to have a platform with less cost and energy consumption.

7.2.2 Average case analysis

For an average case analysis of the system, we have assumed that the loads of all tasks variate according to a uniform distribution, based on the inspiration got from measurements of similar systems. As the UML diagrams provide only the worst case value of the load of each task, we have considered that the actual load varies between 75% and 100% of the value provided. The limits of the load variation for each task are given in fig. 7. Based on

Table 5. Architecture A worst case end-to-end delays

Measured scenario	Other active scenario	POOSL [*ms*]	MPA [*ms*]	UPPAAL [*ms*]
ChangeVolume	HandleTMC	41.771	42.2424	41.796
HandleTMC	ChangeVolume	357.81	390.086	381.632
ChangeAddr	HandleTMC	78.89	84.066	79.075
HandleTMC	ChangeAddr	171.77	265.849	172.106

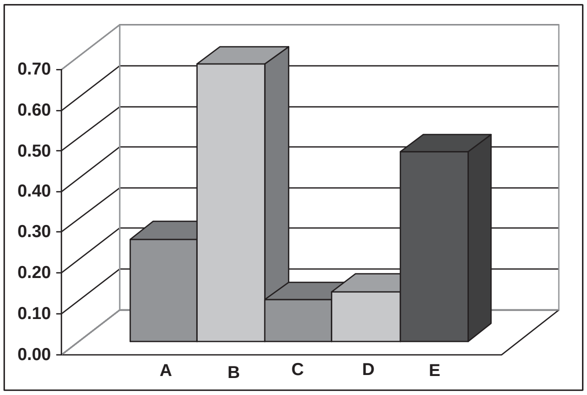


Figure 23. Maximum end-to-end delay for scenario HandleTMC

the MIPS rate of the CPUs on the proposed architectures, given in fig. 22, we can compute the execution times of tasks.

During simulations[6] of the system behaviour for each of the architectures proposed in fig. 22, the end-to-end delays were monitored. The results obtained were graphically plotted as distribution histograms, showing on the horizontal axis the values of the end-to-end delay and on the vertical axis the rate of occurrence of each value. As the parallel execution of two scenarios is likely to lead to more variation in the end-to-end delay, fig. 24 shows the distribution histogram for the HandleTMC scenario when it runs in parallel with ChangeVolume on architecture A. From such distribution histograms, the minimum (best case) and the maximum (worst case) values for the end-to-end delays can be deduced. Columns 3 and 4 in table 8 show these values for all the combinations of scenarios running on architecture A. Moreover, the relative frequency of occurrence of the maximum value can also be deduced. During simulations, we have observed that the requirements are met for all the scenarios on all the proposed architectures and that the maximum delays are much smaller than the deadlines.

7.2.3 Dimensioning of the system

The in-car navigation system is a soft real-time system that allows a rate of 5% of deadline misses. Based on this, together with the utilisation rates of the resources, which were also monitored during simulation, and the observed maximum values of the delays, one can reason about possible platform performance reduction in order to reduce cost and energy consumption of the system.

In [25], where this case study was analysed using MPA, the authors investigated the robustness of architecture A. Therefore, in this paper we have also focussed on this architecture to reason about its resources. The utilisation

[6] By using the fast execution engine Rotalumis, a few minutes of system simulation represent several hours of runtime behaviour. The simulation was run until an accuracy of 99% of the results was reached.

Table 6. Resources utilisations in architecture A

Scenario ChangeVolume	Scenario ChangeAddr	Scenario HandleTMC	MMI [%]	NAV [%]	RAD [%]	Bus [%]
YES	NO	NO	87	0	30	3
NO	YES	NO	3	5	0	1
NO	NO	YES	1	2	4	1
YES	NO	YES	88	2	33	4
NO	YES	YES	4	6	2	2

Table 7. Tasks loads for the average case analysis

Task name	Min [*instr.*]	Max [*instr.*]
HandleKeyPress	7.5E4	1E5
AdjustVolume	7.5E4	1E5
UpdateScreen	3.75E5	5E5
DatabaseLookup	3.75E6	5E6
ReceiveTMC	7.5E5	1E6
DecodeTMC	3.75E6	5E6

of **MMI** is 88%. As the periods and loads of the tasks mapped on this processor are quite heavy, there is not much room for the decrease of its capacity. The **NAV** processor is used 6%. The histograms of scenarios ChangeAddr and HandleTMC showed a difference of 80ms and 200ms respectively between the worst case delays obtained and the requirements. Hence, we reduced **NAV** capacity to 40MIPS. The utilisation of **RAD** is 33%. The analysis showed a difference of 100ms for ChangeAddr and 200ms for HandleTMC respectively between the maximum delays and the deadlines. As there is potential for capacity reduction, we reduce the capacity of this processor to 5MIPS.

With this new configuration for architecture A, we resumed our simulations using the same variances in the task loads and the same task priorities. The distribution histograms of the end-to-end delays were plotted and, as an example, fig. 25 shows the histogram for the HandleTMC scenario. The mean and maximum values of the end-to-end delays for all the scenarios are presented in columns 5 and 6 in table 8. From the confidence intervals calculated during simulation, we observed that the rate of deadline misses is within 5%, thereby fulfilling the requirements. In this way, we have found a better dimensioning of the system than what was found using MPA, reducing two of the processors with 65% (NAV) and respectively 55% (RAD).

Furthermore, in order to use such analysis results in an multi-disciplinary model of complex systems aiming at design trade-offs across disciplines, an abstraction of the timing behaviour of the software part is needed. To this end, we propose to fit the resulting distribution curves into known types of distribution. According to the *central limit theorem* in probability theory, due to the uniformly distributed loads of the tasks and to the fact that tasks in different scenarios are independent, the end-to-end delay of a scenario has approximately a normal distribution. Therefore, over the distribution histogram obtained from a simulation, a normal distribution curve is fitted. Fig. 25 shows such a curve fitted over the HandleTMC histogram. The parameters of the normal distribution are the mean value (μ) of 838.32 (ms) (the mean value of the delay) and the standard deviation (σ^2) of 3953.36 (ms). From such curves, the rate of deadline misses can be deduced, based on their characteristics. For example, the deadline for HandleTMC, which is 1000ms, can be found between two and three standard deviations from the mean. Thus, the probability of missing the deadline is less than 5%, which means the requirements are met. Furthermore, from these curves the probability of rare events occurrence can also be computed.

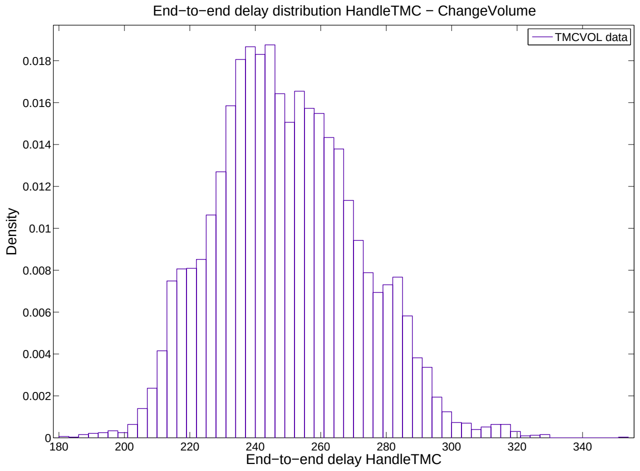


Figure 24. HandleTMC distribution histogram on architecture A

Table 8. End-to-end delays of all scenarios

Measured scenario	Active scenario	Min. delay [*ms*]	Max. delay [*ms*]	Mean delay [*ms*]	Max. delay [*ms*]
ChangeVolume	HandleTMC	28.17	47.82	49.66	58.48
HandleTMC	ChangeVolume	180.9	353.51	838.32	1056.06
ChangeAddr	HandleTMC	61.08	127.51	134.12	270.8
HandleTMC	ChangeAddr	132.59	204.06	349.712	496.03

The analysis approach we considered for the in-car navigation case study is summarised in fig. 26 in which the steps to be performed for the analysis of a soft real-time system are provided.

8. Conclusions

In this paper, we have presented modelling patterns, based on the concepts of the Parallel Object-Oriented Specification Language, for the design space exploration of real-time embedded systems. These patterns allow easy composition of system models consisting of real-time tasks, computation and communication resources and their associated schedulers. Due to the expressiveness of POOSL, important aspects, like task activation latencies and context switches, can be taken into account, enabling the building of realistic models without sacrificing their conciseness. Moreover, due to this reason, the analysis can provide more realistic results than the classical scheduling techniques can.

The use of the patterns presented in this paper reduces both the modelling and the analysis effort. The models made can be analysed for worst-case and average loads, missing of deadlines and deadlock absence. Although completeness cannot be claimed, the efficiency of the model simulation allows exploration of a substantial part of the design space. Furthermore, we presented a way to make an abstraction of the analysis results of the timing behaviour to use it as input for multi-disciplinary models.

As future work, we aim at extending the modelling patterns to cover for complex platforms like networks-on-chip, by taking into account memory components, routing algorithms and even batteries for the analysis of energy consumption.

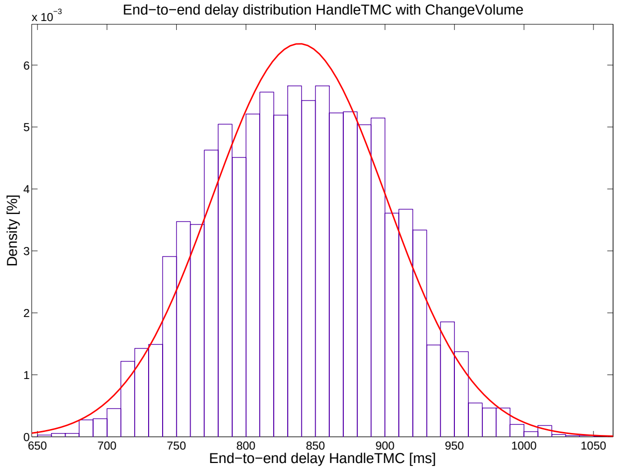


Figure 25. Distribution fitted over the HandleTMC distribution histogram on the improved A

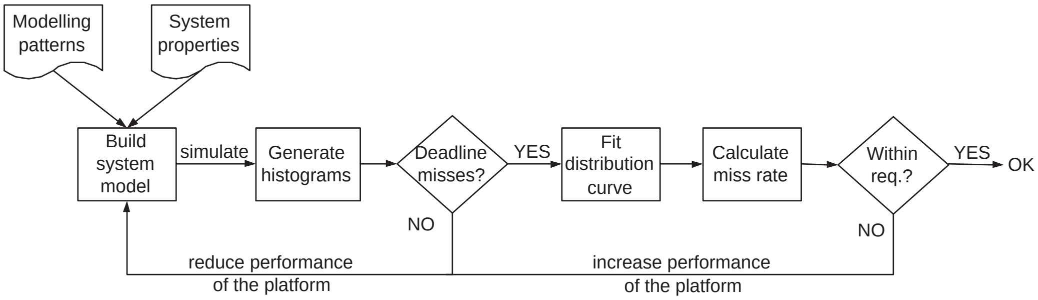


Figure 26. Flow of the steps in the analysis approach

Acknowledgments. The authors would like to thank Marcel Verhoef and Menno de Hoon for their help with the in-car navigation system case study.

References

[1] Modular Performance Analysis. http://www.mpa.ethz.ch/.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 (2), April 1994.

[3] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30 (5): pp. 295–310, 2004.

[4] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A Tutorial on UPPAAL. In: *SFM*, pp. 200–236, 2004.

[5] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe Buttazzo. A Hyperbolic Bound for the Rate Monotonic Algorithm. In: *IEEE Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 59–66, Delft, The Nederlands, June 2001.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[7] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In: *Proc. of the conference on Design, Automation and Test in Europe*. IEEE Computer Society, Washington, DC, USA, 2003.

[8] Bruce Powell Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] Marc G.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Eindhoven University of Technology, 2002.

[11] M. Gries, J. Janneck, and M. Naedele. Reusing design experience for petri nets through patterns. In: *Proc. of High Performance Computing 1999*, 1999.

[12] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration*, 38 (2): pp. 131–183, 2004.

[13] Gilles Kahn. The semantics of simple language for parallel programming. In: *Proc. of IFIP Congress*, 1974.

[14] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In: *Proceedings of the IEEE ASAP*, 1997.

[15] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94 (1): pp. 1–28, 1991.

[16] Paul Lieverse, Pieter van der Wolf, Kees Vissers, and Ed Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *VLSI Signal Processing Systems*, 29 (3): pp. 197–207, 2001.

[17] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the Association for Computing Machinery*, 20 (1), 1973.

[18] Sorin Manolache. *Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour*. PhD thesis, Linkpings University, 2005.

[19] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[20] OMG. *Unified Modeling Language (UML) - Version 1.5*. OMG document formal/2003-03-01, Needham MA, 2003.

[21] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with Artemis. *Computer*, 34 (11): pp. 57–63, 2001.

[22] POOSL. http://www.es.ele.tue.nl/poosl.

[23] Bart D. Theelen. *Performance Modelling for System-Level Design*. PhD thesis, Eindhoven University of Technology, 2004.

[24] Piet H.A. van der Putten and Jeroen P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, 1997.

[25] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using Modular Performance Analysis - A case study. Accepted for publication in the STTT Journal.